SOFTWARE MAINTENANCE: RESEARCH AND PRACTICE, VOL. 9, 127–159 (1997)

Practice

Development and Application of an Automated Source Code Maintainability Index

KURT D. WELKER,[1]* PAUL W. OMAN[2] AND GERALD G. ATKINSON[2]

[1] *Lockheed Martin Idaho Technologies Company, P.O. Box 1625, Idaho Falls, ID 83415 USA*
[2]*Department of Computer Science, Universtiy of Idaho, Moscow, ID 83843, U.S.A.*

SUMMARY

When software maintenance results in changing the source code, the change frequently occurs in an undisciplined manner that results in source code that is inherently more difficult to maintain. The long-term effect may be thought of as a downward spiral that culminates in virtually 'unmaintainable' code where it is more cost effective to just start again. Too often, software personnel and managers, aware that the code is becoming less maintainable, have been unable to estimate with useful accuracy the degree to which maintainability has diminished. The result is that they either continue to waste time and money maintaining code which is effectively unmaintainable, or they opt to live with what seems to be bad code, not touching it for fear of breaking it. To avoid this state of affairs, software developers need to be able to quanitify both the current level of software maintainability and the impact on it of any given change.

This paper discusses the application of software metrics as a tool for quantifying code maintainability and, by extension, for making decisions. The first part of the discussion focuses on deriving a minimal set of easily calculated metrics which, when taken together, can produce a single-valued quantification (or index) of code maintainability. Case studies are then presented which serve to illustrate not only the degree to which software can degrade over time, but how this 'maintainability index' (MI) can be used to quantify maintainability and aid decision making. Finally, a methodology for making metric assessments readily available to software personnel is presented so that they can now easily integrate maintainability metrics into maintenance (and development) processes. As a result, the downward spiral of code degradation becomes at least recognizable, hopefully less precipitous, and perhaps avoidable entirely.

J. Softw. Maint., **9**, 127–159 (1997)

No. of Figures: 12. No. of Tables: 10. No. of References: 40.

KEY WORDS: software metrics; source code degradation; maintainability index; software quality assurance; re-engineering; maintainability case studies

* Correspondence to: Kurt D. Welker, Lockheed Martin Idaho Technologies Company, P.O. Box 1625, Idaho Falls, ID 83415, U.S.A. E-mail: wdk@inel.gov

Contract grant sponsor: The United States Air Force Information Warfare Center
Contract grant sponsor: The Idaho National Engineering Laboratory; Contract grant number: DE-AC07-94ID13223
Contract grant sponsor: The Universtiy of Idaho Software Engineering Test Laboratory

CCC 1040–550X/97/030127–33$17.50

Received 18 November 1996
Revised 1 March 1997

1. INTRODUCTION

Belady and Lehman have documented studies showing that code maintainability decreases as a result of change (Belady and Lehman, 1976; Lehman, 1980). Over time, uncontrolled or undisciplined change activities cause software to become increasingly difficult to maintain. As a result, it is not uncommon to find that 40% to 60% of the cost of production relates to code maintenance (Coupal and Robillard, 1990; Gibson and Senn, 1989; Kafura and Reddy, 1987; Booch, 1987, pp. 3–5, 554–555; Harrison *et al.*, 1982). The huge costs associated with code maintenance have given rise to a search for more efficient ways of implementing the maintenance process. To make the process more efficient, better information is required regarding the degree of difficulty associated with maintaining a particular piece of code to enable better decision making. The key to gathering this information is the ability to identify and measure the attributes of maintainabilty.

Historically speaking, software practitioners interested in measuring software maintainability have struggled in their attempt to gather meaningful metrics. Many traditional software metrics have come under academic scrutiny, and with just cause (Hamer and Frewin, 1993; Curtis, Sheppard and Milliman, 1993; Shepperd, 1993; Basili and Perricone, 1993). However, while many of the objections to traditional metrics are valid, there are significant industrial data to support the contention that the application of rigorously developed metrics can provide useful information to software developers and maintainers (Khoshgoftaar and Oman, 1994; Lowther, 1993). As a result of a research initiative sponsored by Hewlett-Packard and the Idaho National Engineering Laboratory (INEL), the University of Idaho Software Engineering Test Laboratory (SETL) has developed a composite software metric for measuring maintainability which has been implemented with interesting and practical results.

In this paper, we first describe the origin of a single-valued quantification (or index) of maintainability. The intent of this description is to provide the reader with background information on the development and application of our model of source code is change, which we term the maintainability index (MI). A summary of the validation of this metric through industrial applications is also included. Next, four new case studies demonstrate the ease and effectiveness of incorporating maintainability metrics into the maintenance process. Finally a methodology for making metric assessments readily available to engineers is presented, which should enable software practitioners to easily incorporate maintainability metrics such as the MI into their maintenance processes.

We do not claim that the the MI is the only model for predicting maintainability, nor do we claim it is the best overall. Different models could be constructed, such as the maintainability index (MI) advanced by the CCTA in Great Britain (West, 1993; Peercy, 1996). Clearly, more accurate—and complicated—maintainability models can be constructed than our MI. Rather, we have deliberately kept the simplicity of our model to that which can be calculated using existing metrics and a hand-held calculator. This was done to provide the working software practitioner with a quick, easy and reasonably accurate method of modelling the maintainability of source code, a method easily

implemented using the automated techniques described in this paper. Additionally, the MI model has shown a remarkably consistent fit with software practitioners' intuitive judgements. Because of these two criteria, we have found that the MI is a practical tool for assisting software practioners and managers in gathering better data and thereby making better code maintenance decisions (Welker and Oman, 1995).

2. MAINTAINABILITY INDEX (MI)

2.1. MI introduction

The MI is a single-valued, composite metric that quantifies the maintainability of a software product. It was developed through a series of experiments sponsored by Hewlett-Packard under grant and contract numbers HP-1720063 and HP-5688. The SETL was charged with developing a quick and relatively simple method for determining the maintainability of program source code. Therefore, the MI was constructed such that it took advantage of existing tools and could be calculated at each software practitioner's desktop. This section describes the original research efforts, the evolution of the MI metric, and the subsequent validation in industry across numerous applications in a variety of environments and organizations.*

2.2. Constructing the MI

As a starting point, we used the IEEE definition of software maintainability:

'The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.' (IEEE, 1990)

While this facilitated understanding of the goal, it quickly became apparent that quantifiable measures would not be forthcoming from this definition alone. The next step, therefore, was to broaden our understanding of the nature of maintainability. Definitions from 35 published works on software maintainability were examined, and 92 maintainability attributes were identifed (Oman, Hagemeister and Ash, 1991). For attributes that did not already possess measurement criteria, metrics were developed. Eventually we were able to show how each of these attributes could be measured via a battery of 62 metrics.

Since a sub goal of the research effort was to identify a minimal set of easily computed metrics, our experimentation initially focused on determining which metrics were most useful. Metrics which were difficult to compute were eliminated, such as those requiring historical or subjective data. As a result, the number of metrics was cut to 40. Experimentation then began in order to determine how the various metrics could be combined into

*Note: Readers already familiar with the development and application of the MI are invited to skip to Section 3 of this paper.

a single-valued quantification or 'index' of code maintainability (Oman and Hagemeister, 1994).

A suite of eight programs was obtained from Hewlett-Packard. The programs ranged in size from 1 000 to 10 000 lines of code, and subjective assessments of the maintainability of the software were obtained from the software practitioners in charge of maintaining the code. The software practitioners, who were unaware of the purpose and structure of the study, were surveyed to obtain an expert's assessment or 'maintainability rating' for each set of code. A range from 25 to 125 was used. The instrument used to gather the subjective data was constructed from the U.S. Air Force software maintainability evaluation instrument (AFOTEC, 1989). Readers interested in the construction of this instrument should see (Oman and Hagemeister, 1994). The purpose of the subjective evaluations was to provide numerical ratings which could be used as dependent variables for regression analyses.

Using the 40 metrics identified earlier, a series of approximately 50 regression tests were then conducted in an attempted to identify simple models which could be applied to a wide range of software. Metrics were calculated for each subroutine within the eight code suites. Using these metrics as indicators of maintainability inherent in each of the code samples, a series of correlations and principal component analyses were conducted in which the measured data were compared against the subjective evaluation. The analyses were used to determine which metrics were computationally redundant with other metrics in the set and which metrics were leading indicators of maintainability as measured by the initial survey. The leading indicators became candidates for inclusion in the maintainability model.

From the resulting correlations and the principal component analyses it was apparent that:

(1) The Halstead (1977) metrics for program effort (E) and volume (V) were very strong predictors of maintainability.
(2) The two cyclomatic complexity metrics, the McCabe cyclomatic complexity ($V(g)$) (McCabe, 1976) and the Myers extended cyclomatic complexity ($V(g')$) (Myers, 1977), were equally significant predictors of maintainability. Either can be used for modelling maintainability; it does not matter which. We chose to use the extended cyclomatic complexity measurement, hereafter referred to as $VG2$.
(3) The number of lines of code (LOC) and number of lines of comments (CMT) contributed significantly to the prediction of maintainability. (Note: a line of code is defined as a physical line of source code, and a line of comment is a physical line on which there is a comment.)
(4) Subroutine averages, rather than total metric values, tend to be more stable and, therefore, more indicative of the true nature of the maintainability of a software suite. This paper uses the term 'subroutine' or 'module' for any named lexical component of a program, which given a specific programming language might be a function, a procedure, a subroutine, a section, a module, etc.

In spite of the current research trend away from the use of Halstead's metrics, all of our tests clearly indicated that they were the best predictors of maintainability of our test data. The value of Halstead's metrics—particulary effort and volume—exceeded all other

metrics in terms of their ability to assess or predict maintainability. Two other studies by independent researchers have found similar results. In a study of code metrics, structure metrics and hybrid metrics, Wake and Henry (1988) found that Halstead's effort was the best single predictor of maintainability as measured by the number of lines of code changed. Also, a study by Harrison (1988) found that Halstead's effort was the best single predictor for test resource allocation. Altogether, these three studies provide evidence that Halstead's hybrid measures (i.e., effort and volume) are quite useful when assessing or predicting maintainability.

Although we tested all 40 of the metrics, our efforts showed that models using Halstead's effort, lines of code, extended cyclomatic complexity and lines of comments were best. As a result of our analysis, three potentially useful software maintainability assessment models were constructed. These three models are listed below, in order of increasing complexity. Note that as the models become more sophisticated, the degree of variance accounted for by the model, as expressed by the R^2 statistic, increases. The R^2 value indicates the proportion of variation within the data set which is explained by the model. Thus, an R^2 value of 1·0 is perfect, while R^2 values greater than 0·80 are considered to be very good. Although we derived models with R^2 values exceeding 0·98, it was clear from the make-up of those models that they were specific to the test data set and would not be applicable to a wider range of software.

As the models become more sophisticated, each successive maintainability metric does a better job of modelling the subjective maintainability assessment obtained from the software practitioners. Both principal components analyses and factor analyses were conducted on the metrics set for the original Hewlett-Packard data used to construct the MI polynomials, and the separate data sets used in the initial validation tests of the models. Although studies have shown that metrics such as LOC and $V(g)$ can be highly colinear, our analyses put LOC and $V(g)$ in separate measurement domains. Furthermore, no single metric or combination of two metrics provided as powerful a predictive model as that obtained through the use of LOC, $V(g)$ and volume combined. Details of these analyses can be found in Oman and Hagemeister (1994), Zhuo (1992) and Zhuo *et al*. (1993).

(1) *The original single-metric MI model*:

$$\text{Maintainability index} = 125 - 10 \times \log(\text{aveE}) \tag{1}$$

where aveE is the average Halstead effort per module. The R^2 for this original single-metric maintainability index mode is 0·72.

(2) *The original four-metric MI model*:

$$\text{Maintainablity index} = 171 - 3{\cdot}42 \times \ln(\text{aveE}) - 0{\cdot}23 \times \text{ave}VG2 - 16{\cdot}2 \times \ln(\text{aveLOC}) + 0{\cdot}99 \times \text{aveCMT} \tag{2}$$

where: aveE is the average Halstead effort per module, ave$VG2$ is the average extended cyclomatic complexity per module, aveLOC is the average lines of code per module, and aveCMT is the average number of lines of comments per module. The R^2 for this original four-metric maintainability index model is 0·90.

(3) *The original five metric MI model*:

$$\text{Maintainability index} = 138 - 2{\cdot}76 \times \ln(\text{aveE}) - 0{\cdot}33 \times \text{ave}VG2 - 12{\cdot}2 \times \ln(\text{aveLOC}) + 0{\cdot}88 \times \text{aveCMT} + 1{\cdot}04 \times \text{EDOC} \quad (3)$$

where aveE is the average Halstead effort per module, ave*VG*2 is the average extended cyclomatic complexity per module, aveLOC is the average lines of code per module, aveCMT is the average number of lines of comments per module, and EDOC is a 20 point subjective evaluation: 15 points for external documentation and 5 points for the ease of building the system. The R^2 for this five-metric maintainability index model is 0·94.

The EDOC metric in the five-metric model was added in an attempt to better approximate the maintainability assessments obtained from the software practitioner survey. The initial survey contained subjective queries about the external documentation and development processes of the software being assessed. These aspects of software maintainability can never be adequately captured by analysing source code alone, so a single subjective metric (EDOC) was added to the polynomial to account for external documentation and ease of building the system.

2.3. Initial validation

In order to test the reliability and applicability of our maintainability models, a new set of source code was acquired. Six more programs (ranging in size from 1 000 to 8 000 lines of code each) were obtained from Hewlett-Packard along with subjective engineering assessments of the maintainability of the code. This time, the instrument used to gather the subjective data was constructed directly from the U.S. Air Force software maintainability evaluation instrument (AFOTEC, 1989) without rewording either the questions or the response format (Oman and Hagemeister, 1994).

The single, four and five-metric models were used to calculate MIs for the six new programs. A statistical analysis was then conducted to compare the calculated MIs with the survey results. Even though a different maintainability assessment survey was used to quantify the experts subjective assessments of the software system, all three of the MI models did a good job of predicting software maintainability at the system and file levels. R^2 values for the initial validation test ranged from 0·74 to 0·89, depending upon the model used (Oman and Hagemeister, 1994).

2.4. Model evolution

While the inital models did appear to be good predictors of maintainability, as compared with the experts' assessments, there were three obvious problems. First, the five-metric model, by including the EDOC metric, became more subjective and difficult to use. Second, both the four and five-metric models appeared to have the potential to be overly sensitive to comments. This was confirmed by application to other industrial systems (Lowther, 1993). Finally, and despite our results (as well as the those of Wake and

Henry (1988) and Harrison (1988)), some opinion does not place much 'good faith' in Halstead's effort metric. While divergent opinion would not influence the performance of the proposed models, the concern was that some software developers who might have otherwise derived great benefit from acceptance and use of the model might be overly skeptical.

To combat these shortcomings, the MI was fine-tuned over time. As more data became available, from varying industrial sources, a greater degree of confidence developed in the fit of certain metrics. As a result, the original single, four and five-metric models gave way to three and four-metric MI models.

The single-metric MI model was abandoned for models which simply had better fit, and the five-metric model was dropped in order to eliminate the subjectivity introduced by the EDOC factor. The influence of comments was dealt with in two ways. First, a three-metric MI was developed simply by dropping the comment factor from the original four-metric model. The four-metric MI was modified to include a cap on the amount of influence comments could exert.

The final perceived shortcoming of the original models to be dealt with was the contention that Halstead's effort was not as good a predictor as Halstead's volume. There has been much discussion of the non-monotonicity of Halstead's effort metric, i.e., it is not a non-decreasing function under the concatenation operation. The original experimentation had shown that, and that both effort and volume were excellent indicators of maintainability. Since the difference between the two was not statistically significant, we reconstructed the model using Halstead's volume metric (Coleman *et al*., 1994). Definitions of the improved models follow:

(1) *The improved, three-metric MI model* (presently being used in our ongoing studies):

$$\text{Maintainability index} = 171 - 5{\cdot}2 \times \ln(\text{aveV}) - 0{\cdot}23 \times \text{ave}VG2 - 16{\cdot}2 \times \ln(\text{aveLOC}) \tag{4}$$

where: aveV is the average Halstead volume per module, ave$VG2$ is the average extended cyclomatic complexity per module, and aveLOC is the average lines of code per module.

(2) *The improved, four-metric MI model* (also presently being used in our ongoing studies):

$$\text{Maintainability index} = 171 - 5{\cdot}2 \times \ln(\text{aveV}) - 0{\cdot}23 \times \text{ave}VG2 - 16{\cdot}2 \times \ln(\text{aveLOC}) + 50 \times \sin(\sqrt{(2{\cdot}4 \times \text{perCM})}) \tag{5}$$

where: aveV is the average Halstead volume per module, ave$VG2$ is the average extended cyclomatic complexity per module, aveLOC is the average lines of code per module, and perCM is the average per cent of lines of comments per module.

In order to determine which expression best represents the maintainability of a given software system, some human anlaysis of the comments in the code must be performed. If any of the following criteria are true, then the three-metric MI may be a better fit than the four-metric MI for measuring maintainability:

- If the comments do not accurately match the code. It has been said 'the truth is in the code'. Unless considerable attention is paid to comments, they can become out of synchronization with the code and thereby make the code less maintainable. The comments could be so far off as to be of dubious value.
- If there are large, company standard comment header blocks, copyrights and disclaimers. These types of comments provide minimal benefit to software maintainability; as such, the four-metric MI will be inflated, and provide an overly optimistic maintainability picture.
- If there are large sections of code which have been commented out. Code which has been commented out creates maintenance difficulties.

Generally speaking, if it is believed that the comments in the code significantly contribute to maintainability, then the four-metric MI is the best choice. Otherwise, the three-metric MI will be more appropriate.

2.5. Subsequent validation

Over the last four years, the improved versions of the MIs have been used repeatedly and successfully to assist software practitioners and managers in quantifying the maintainability of industrial system code. While there are recognized theoretical objections to the MI (the question of creating a weighted combination of different metrics of different scales, for instance), the reason for its continued use is that in every empirical test of MI, the results have been confirmed or validated by experienced software practitioners working on the code under study. In every case, subjective evaluations by the software personnel in charge of maintaining the code have closely matched those predicted by the MI. A software maintainability model is only useful if it can provide real world developers and maintainers with more information about the state of the software. Hence, the data used to test and validate our models consisted entirely of genuine industrial systems. Following is a list of MI validation studies conducted on industrial code:

- An analysis of a new HP proprietary system comprising over 200 000 LOC confirmed the predicted high quality of the tightly controlled development effort (Coleman *et al*., 1994).
- A similar analysis of over 200 000 LOC of third party code was used to support a buy/build decision (Coleman *et al*., 1994).
- A pre/post analysis of a small HP subsystem re-engineering effort demonstrated improved maintainability as measured by MI (Coleman, Lowther and Oman, 1995).
- An MI comparison of four systems each greater than 100 LOC corresponded to actual maintenance histories (Coleman, Lowther and Oman, 1995).
- An experiment identifying problem modules in a 236 000 LOC 'C' software system proved that MI was useful in targeting code 'hot spots' (Coleman, Lowther and Oman, 1995).
- A comparison of MI with recorded defects for 29 system components showed a correlation of +0·83, indicating the strong relationship between faults and maintainability (Coleman, Lowther and Oman, 1995; Oman, 1995).

- An analysis of a small public domain software tool demonstrated how MI could be used to monitor and control code quality (Ash *et al*., 1994).
- A rank-ordering of actual and predicted component quality showed a correlation of +0·81 between software practitioner assessments and MI for 37 software components (Oman, 1995).
- An analysis of four different maintenance activities on Hewlett-Packard proprietary code demonstrated how MI could be used to gauge the quality and effectiveness of code change (Pearse and Oman, 1995).

Hewlett-Packard, however, has not been the only venue for validation. The MI has also proven to be accurate and useful in other industrial settings as well as in the governmental arena. Recent work with Northern TeleCom again confirmed the efficacy of the MI as a predictor of degrading maintainability, as has work with the Department of Defense (including various experiments for the U.S. Air Force) and the Department of Energy (including the INEL case studies presented later in this paper).

Empirical testing of the improved four-metric MI model has suggested two cut-off points. MI values below 65 correspond well with experts' evaluations of low maintainability, and values above 85 correspond well with high maintainability. The range of 65 to 85, inclusive, fits nicely with moderate maintainability (Coleman, 1992). Where not explicitly stated in Section 3, these cut-offs may be inferred.

While this clearly does not constitute exhaustive testing in all possible environments, it does show that the MI has applicability beyond academia, and that it can be advantageous in many industrial settings. Notable use of the MI has been achieved in systems designed with various paradigms and written in numerous programming languages with no apparent bias towards any particular design method or implementation language. We now turn to several examples of successful application of the improved three and four-metric MIs.

3. MI CASE STUDIES

3.1. Introduction to case studies

As previously noted, code maintainability decreases over time as a result of change (Belady and Lehman, 976; Lehman, 1980). Frequently, change, regardless of the type of change, occurs without regard to the effect that it has on the maintainability of the code; the only concern is whether or not it can survive the (regression) test suite. As a result, it often becomes increasingly difficult to make changes in the future. The long-term effect may be thought of as a downward spiral that culminates in a state in which the code becomes virtually unmaintainable, where it is more cost effective to just start again. We contend that this outcome is at least postponable, and perhaps avoidable altogether, when information concerning the state of code maintainability is readily available to software practitioners and managers.

From a software engineering perspective, the problem is twofold. First, we must discover a means of measuring the impact of change; this we addressed earlier in Section 2.2. Second, the maintenance process itself must be modified such that quantification of

maintainability is both easy to calculate and readily available to software practitioners and managers. To this end, Section 4 illustrates how automated tools can be easily constructed such that they are available at the push of a button. The present Section uses case studies to illustrate how one organization has attacked the problem using the MI as the vehicle for measuring maintainability.

However, it is not the MI itself that is important, but the fact that metrics are used in the maintenance cycle. The main point of the case studies presented here is that periodic measurements can and have been used to improve both software products and software processes. The MI is just one tool for accomplishing such assessments. The case studies use an older, effort-based variant of the MI models. Studies have shown less than two per cent variation between the results of the effort-based MI models and the volume-based models. The respective three and four-metric MI models used in the case studies are variants of models (4) and (5) as follows:

Three-metric MI used

$$\text{Maintainability index} = 171 - 3{\cdot}42 \times \ln(\text{aveE}) - 0{\cdot}23 \times \text{ave}VG2 - 16{\cdot}2 \times \ln(\text{aveLOC}) \tag{6}$$

Four-metric MI used

$$\text{Maintainability index} = 171 - 3{\cdot}2 \times \ln(\text{aveE}) - 0{\cdot}23 \times \text{ave}VG2 - 16{\cdot}2 \times \ln(\text{aveLOC}) + 50 \times \sin(\sqrt{(2{\cdot}4 \times \text{perCM})}) \tag{7}$$

The case studies presented here are:

Case Study 1: an examination of how source code that had degraded over a five-year period could be quantified against an earlier baseline.

Case Study 2: an examination of the quantification of source code maintainability for code that had continued to degrade despite translation to a more 'modern' language, as compared with code which was completely re-engineered.

Case Study 3: an examination of how the integration of an automated maintainability assessment aided making decisions aimed at perfective maintenance.

Case Study 4: an examination of the applicability of maintainability assessments to object-oriented code.

3.2. Case study 1

As part of an INEL-sponsored software metrics research study, a 220 KLOC, proprietary, legacy software system was assessed. It was believed that a metrics analysis of this system would provide insight into the structural changes of the system over the maintenance life cycle. Though the software system was over 20 years old, source code was only available for the past five years. As part of the analysis, two baselines were analysed,

the current baseline and a baseline from five years ago. The older baseline will be referred to as baseline X and the current baseline will be referred to as baseline X^C.

This case study illustrates two main points. First, this study supports the contention that code maintainability does in fact decrease over time, though the root cause of the decrease may not be readily apparent. Second, this study shows how the MI was used to quantify the degree to which code maintainability decreased.

As the first part of our analysis, low-level metrics (such as subroutine counts) were obtained for the two baselines. In this case UX-Metric (see Section 4), an automated software metrics gathering tool, was used to easily and rapidly generate the metrics from the source code. As mentioned in Section 2.2, subroutine averages, rather than total metric values, tend to be more stable and therefore more indicative of the true nature of the maintainability of the software suite. For this reason, these values were calculated from the low-level metrics, also using UX-Metric. Finally an INEL metrics toolset (see Section 4) was used to synthesize the MI values from the low-level metrics; initially both the three-metric and four-metric MIs were computed for the system. Table 1 summarizes the key data.

A quick comparison of the data from the two baselines indicated that both the size and complexity of the code had increased over time. Since these two metrics are leading indicators of maintainability and are integral to the MI, it was expected that the MI would show a decrease in code maintainability, as it did. However, the MI values provided two additional, interesting insights. First, over the past five years, the MI for the entire system has dropped from 26·3 to 18·2 and from 60·6 to 52·0, for the three and four-metric MIs respectively. This decline was large enough to be significant, though the maintainability was obviously quite low to begin with.

The other important point derives from the large difference between the values for the three-metric MI, which does not take comments into account, and the four-metric MI, which does. The average difference between the two metrics is approximately 30. For other systems the four-metric MI does typically have a higher value, but the difference is usually not so large. Though the system would be classified as having low maintainability in either case, the difference was abnormal. Obviously, comments were playing an important role in determining the quantification of maintainability. Before we could determine which MI more accurately reflected the state of the system, an inspection of the source code was necessary. A visual inspection of the source code showed large sections of commented-out code, comment headers and blank comment lines, but very few meaningful comments. This indicates that not only was the four-metric MI unduly

Table 1. Low-level metrics, averages, and maintainability indicies

Baseline name	Sub routines	Total LOC	Average LOC	Total effort	Total *VG*2	Average *VG*2	3-metric MI	4-metric MI
X	627	124 425	198·45	2·95E9	17 549	27·99	26·31	60·64
X^C	866	221 636	255·93	7·29E9	31 584	36·47	18·25	52·04

Table 2. Subroutine maintainability categorization

Baseline name	Total number of subroutines	Number of high maintainability	Number of moderate maintainability	Number of low maintainability
X	627	66	156	405
X^C	866	70	168	628

inflating the results, but that the three-metric MI was probably giving a more accurate description of system maintainability.

To better understand the significance of derived MI values, the Hewlett-Packard cut-off points established in Coleman (1992) were overlaid on the data. These cut-off points categorize software with an MI of less than 65 as having low maintainability; MI greater than 85 shows high maintainability, and 65 to 85, inclusive, indicates a moderate level of maintainability. The results of this categorization are shown in Table 2 and Figure 1.

Two important points emerge from these data. First, the percentage of subroutines which is highly maintainable is fairly low for either baseline. Second, the precentage of subroutines of high and moderate maintainability is declining over time, and the percentage of subroutines which are difficult to maintain is increasing. These factors signify overall system degradation.

While a breakdown of maintainability by subroutine sheds more light on the maintainability picture, one of the problems inherent in a composite metric such as the MI is that it tends to hide some of the information contained in the individual metrics. Looking at a breakdown by subroutines only gives a part of the picture of maintenance difficulty. As shown in Coleman, Lowther and Oman (1995), looking at raw metrics, such as LOC, in each of the maintainability categories (low, moderate, high) sometimes presents a more complete picture for the amount of code comprising the subroutines in each category. By totalling the LOC for the subroutines in each category and computing the percentage of that cagegory relative to the whole system we obtained the information contained in Table 3 and Figure 2.

Several important points emerge from these data. First, the largest percentage of the code is contained in subroutines which are classified as difficult to maintain. Second, the preponderance of the code being added to baseline *X* falls into the low maintainability category. The above trends are signs of significant system degradation. We know from

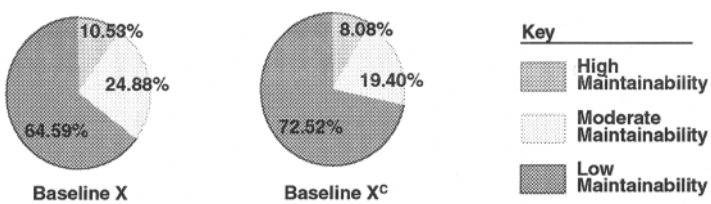

Figure 1. Subroutine maintainability index categorization by percentages

Table 3. LOC per maintainability categorization

Baseline name	Total LOC	High maintainability	Moderate maintainability	Low maintainability
X	124 425	4 196	4 825	115 404
X^C	221 636	1 729	7 993	211 914

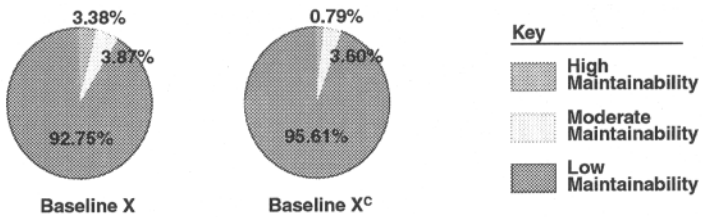

Figure 2. LOC per maintainability categorization by percentages

these analyses that most of the code comprising the system falls in subroutines of low maintainability, and the number of subroutines of moderate and high maintainability is decreasing during the maintenance process.

It is very difficult to establish the root causes for the values of these particular metrics, for there are several possibilities to be considered. One possible cause for the increasing complexity and decreasing maintainability is the increasingly complex nature of the problem domain. Over the years, the application domain technologies have changed, and the need for finer granularity modelling has emerged. As a result, algorithms and code have become more complicated in an effort to appropriately model the technology changes. Other possibilities which could contribute to the development of code which is difficult to maintain are an immature software development process, use of older software technologies and weaknesses in overall software architecture.

Figure 3 shows the MI changes for all of the subroutines which had the same names in both the earlier baseline and the current baseline. The figure presents data as a scatterplot which compares, on a subroutine basis, the three-metric maintainability index levels from both baselines. The vertical axis is the MI; the horizontal axis is a sequence of the baseline X subroutines. Ordering (sorting) those subroutines from low to high MI and then plotting the MI for each subroutine yields then the primary curve. It appears thick and dark from the close spacing of the plot points (+ symbols) for the 520 of the 627 subroutines of baseline X. The * symbols mark the MI plot positions of the baseline X^C subroutines placed horizontally at the same position that each subroutine had in the X baseline. The maintainability levels of 98 subroutines improved over the period, while 402 declined and 20 showed no change. This graphical representation of migration from baseline X to baseline X^C reveals two very interesting phenomena, as denoted by the areas encompassed by the dashed lines.

First, notice the clustering of data points encompassed by the dashed line in the upper right-hand quadrant, which includes most of the subroutines making gains in main-

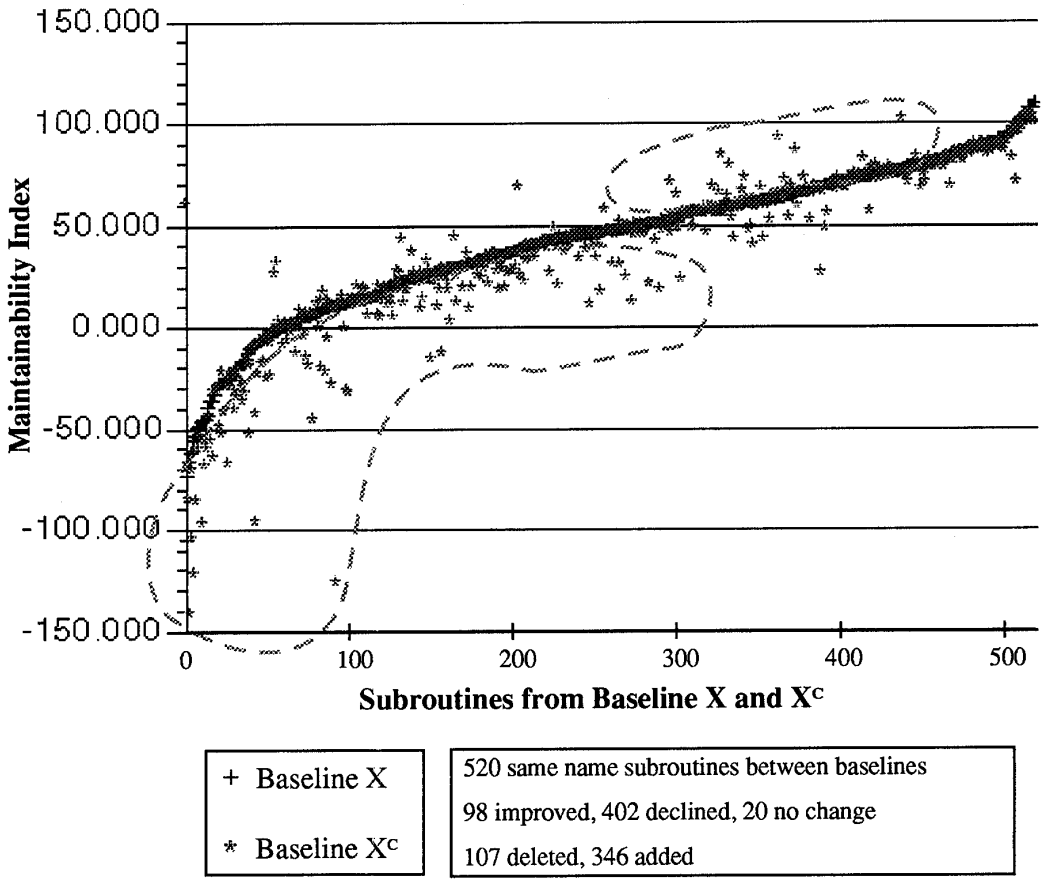

Figure 3. Subroutine MI change over the five-year period

tainability. The subroutines which seem to have made the biggest improvements in maintainability are the ones which originally exhibit high maintainability. While this conclusion is fairly obvious to the observer, and is intuitively believable, a statistical analysis could neither confirm nor deny it (Engelhardt, 1995). Subroutines with lower original MI values seem to have made far fewer gains (as revealed by the relative paucity of subroutines above the baseline in the lower left-hand portion of Figure 3).

Second, Figure 3 also shows that subroutines which were originally difficult to maintain were the most likely to have degraded. A statisitical analysis conclusively showed that this apparent, intuitive conclusion was correct. 402 subroutines from the earlier baseline showed a decrease in maintainability; notice a clustering of data points (encompassed by the dashed line in the lower left-hand quadrant), which highlights notable degradation between baselines. Further, the subroutines whose maintainability decreased over time may be separated into two subsets: 'more maintainable- (three-metric MI > 50) and 'less maintainable' (three-metric MI < 50). The set of 'more maintainable' subroutines degraded

by an average of 4·45 and a median of 2·29. In contrast, the 'less maintainable' subroutines degraded by an average of 10·50 and a median of 5·10 (Engelhardt, 1995).

What is not shown in Figure 3 is that 107 subroutines have been deleted from the earlier 627 subroutines in the *X* baseline and that 346 subroutines have been added. It is also important to note that the previous comparisons did not take into account subroutine name changes. That is, some of the subroutines counted as deleted or added might have been accounted for had name changes been considered. Figure 3 does not show these cases either. Figure 4 gives a more complete view of the maintainability levels of the deleted subroutines, and Figure 5 of the added subroutines. The 107 subroutines deleted typically had higher MI values than the MI values of the 346 subroutines added. In both figures, the subroutines are ordered (sorted) from low (towards the left) to high (towards the right).

A analysis of the MI data shows that the MI of subroutines deleted from the earlier baseline, compared with the MI of those added to become part of the current baseline, is statistically significant, with an average MI of 70·15 for the subroutines deleted versus 43·03 for the subroutines added (Engelhardt, 1995). Less maintainable subroutines are replacing more maintainable subroutines. Though the metrics do not suggest a cause for this, it is quite apparent that the overall system maintainability is decreasing over time.

In summary, Case Study 1 illustrates two main points. First, this study confirmed that code maintainability did decrease over time; the number of subroutines with high maintainability clearly decreased, while the number of subroutines with low maintainability increased significantly. Even though the root causes of the decrease were not apparent from the metrics, the metrics did serve to warn the software practitioners and managers that something undesirable was occurring and that further attention was warranted.

Second, this study showed how metrics (in this case the MI) were used by analysts to

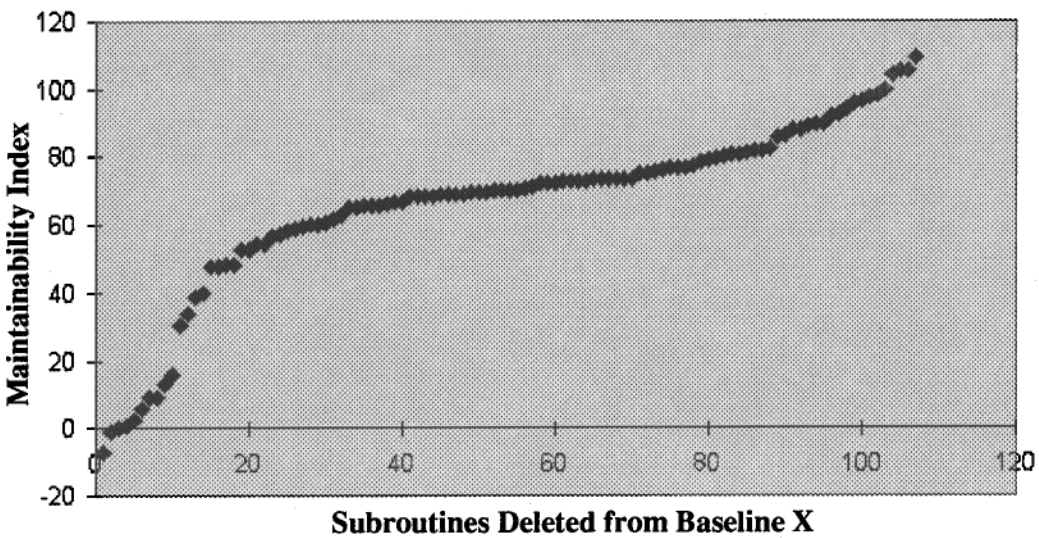

Figure 4. MI of subroutines deleted between baselines

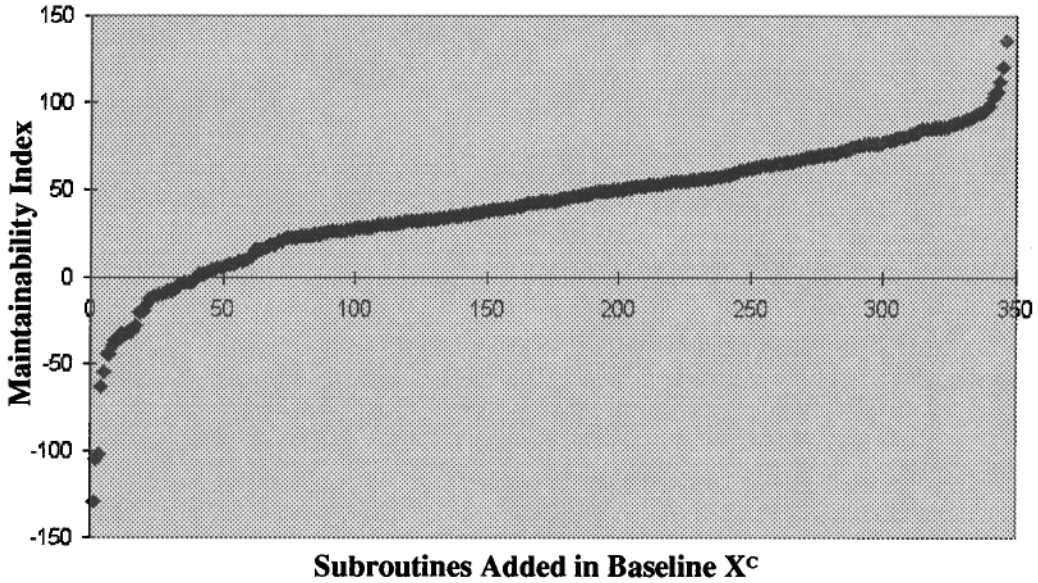

Figure 5. MI of subroutines added between baselines

quantify the degree to which code maintainability decreased as size and complexity increased. Two points of particular interest are that subroutines which are initially highly maintainable tend to remain highly maintainable and that subroutines which were added later tended to have low maintainability. This suggests that if a system could be originally composed of fairly maintainable subroutines, system degradation might be better controlled and even abated. While this conclusion is fairly intuitive, the numerical evidence seems to support it. Application of this knowledge might, theoretically, enable software practitioners to build a system which is not only resistant to code degradation, but which actually becomes more maintainable over time.

As we will see in the next case study, quantification can provide objective support to decision making activities.

3.3. Case study 2

In 1984, the U.S. Air Force Information Warfare Center (AFIWC) developed an in-house electronic combat modelling system known as the 'improved many-on-many' (IMOM). The IMOM software operated in a proprietary hardware/software environment and was written in FORTRAN. The IMOM software became so successful that a large user base was established, and the model's functional capabilities quickly expanded.

In 1989, in an effort to rehost the IMOM software into an open system environment, the Idaho National Engineering Laboratory (INEL) converted the IMOM software by direct translation to the C programming language, and changed hardware and software environments.

Both the FORTRAN and the C baselines of the IMOM software continued to evolve.

In 1991, after conducting an engineering research study which focused on strategic objectives, the INEL re-engineered, by redesigning and completely rewriting, the code for the IMOM software into the Ada programming language, using object-oriented analysis and design.

After the software had been re-engineered, the parties involved with the model believed that the re-engineering effort had been successful in achieving its objectives. The FORTRAN and C baselines of IMOM were retired and the Ada baseline then continued to evolve. As a result of the successful re-engineering of IMOM, the AFIWC directed the INEL to re-engineer three other electronic combat models: a communications jamming model (COMJAM), a reconnaissance model (RECCE), and a passive detection model (PD), all with histories similar to IMOM. Early baselines of these models were in part derived from IMOM model source code.

The AFIWC/INEL team has continued to evolve these models over the past several years. Figure 6 depicts the history of IMOM and the associated electronic combat models.

The software developers claimed that the re-engineered software was indeed maintainable. While this human assessment of the system was certainly valuable, it was possibly biased since, after all, most software is more easily maintained by the developer than by another party.

This case study attempts to answer that question by showing how the MI was used by engineers to quantify the degree to which code maintainability decreased over time as a result of change and the degree to which a re-engineered system's maintainability was improved. Also, as in Case Study 1, this case study serves to confirm the contention that code integrity does in fact decrease over time if no specific efforts are taken to prevent code degradation. Finally, this case study shows how the conduct of periodic assessments of code maintainability were used as a management tool to help decide if it was more appropriate to re-engineer a system than to maintain it.

The IMOM family software baselines were processed using UX-Metric, an automated software metrics gathering tool to gather low-level metrics on the source code at both system and subroutine levels. Next, an INEL metrics toolset (see Section 4) was used to synthesize the MI values from the low-level metrics. Table 4 summarizes the results.

By comparing the metrics from each of the different baselines noticeable trends emerge. These metrics provide a glimpse into the complexities of the IMOM family baselines of software. They show the trend towards increased complexity over time, and they indicate the complexity reduction that occurred with re-engineering. In order to get a fuller picture though, a synthesis of these results which takes into consideration each of the metrics in respect of each other is needed. This is where the MI metric provided a valuable addition to understand the meaning of traditional metrics results.

In an effort to better assess the success of the re-engineering effort from a quantitative maintainability perspective, a three-metric MI was calculated for each baseline of the system (Welker, 1994). In addition, software developers familiar with the source code for the various systems were surveyed using the SETL's Source Code Maintainability Survey (Oman and Hagemeister, 1992) to determine a human measurement which uses a numerical scale that corresponded to MI. Table 4 indicates the three-metric MI for each software baseline analysed. Notice the dramatic improvement achieved by re-engineering, as quantified by the three-metric MI.

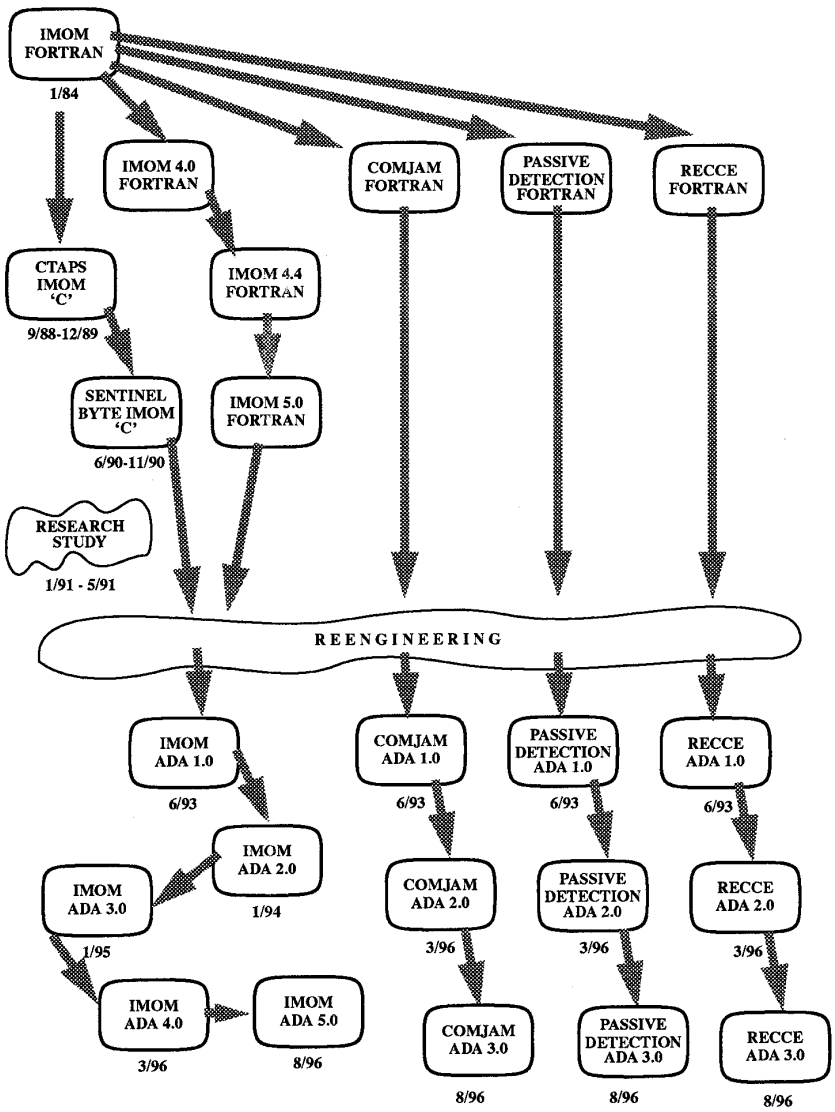

Figure 6. History of IMOM family of electronic combat models

For comparison, subjective human assessments of maintainability were made for three systems. Although the numerical values of the subjective assessments do not correspond exactly to the MI values obtained for those systems, the scale and direction of the results are similar. These survey values are given in the Notes for Table 4. The four-metric MIs have also been included for the current baselines of the systems since it is believed that the comments in the software are significantly meaningful. Unfortunately, the four-metric

Table 4. IMOM family software metics summary

Baseline name	Sub-routines	Total LOC	Average LOC	Total effort	Total *VG*2	Average *VG*2	Three-metric MI	Four-metric MI
F IMOM 4.0	341	83 174	243·9	1·22E8	3 868	11·34	35·6	
F IMOM 4.4	399	106 603	267·2	1·92E8	5 354	13·41	32·6	
F IMOM 5.0	478	128 055	267·9	7·57E8	9 367	19·59	27·1	Note 1
C CTAPS IMOM 1.0	298	75 357	252·9	1·66E8	4 591	15·40	32·5	
C SB IMOM	422	109 182	258·7	2·51E8	6 886	16·31	31·7	Note 2
A IMOM 1.0	1 562	61 300	39·2	5·70E7	5 300	3·39	74·8	
A IMOM 2.0	1 976	76 641	38·8	6·53E7	6 997	3·54	75·3	Note 3
A IMOM 3.0	2 516	119 084	47·3	1·31E8	9 011	3·58	70·5	
A IMOM 4.0	2 506	128 473	51·2	1·31E8	9 204	3·67	69·2	107·9
A IMOM 5.0	2 778	142 197	51·2	1·48E8	10 457	3·76	69·1	107·0
F COMJAM	218	55 279	253·6	6·56E7	2 687	12·32	35·3	
A COMJAM 1.0	1 540	61 230	39·8	5·55E7	5 197	3·37	74·6	
A COMJAM 2.0	1 506	72 109	47·8	5·73E7	4 926	3·27	71·5	110·8
A COMJAM 3.0	1 590	75 974	47·8	5·99E7	5 243	3·29	71·6	110·9
F PD	295	41 645	141·2	5·18E7	2 553	8·65	47·5	
A PD 1.0	1 568	62 867	40·1	5·79E7	5 393	3·43	74·4	
A PD 2.0	1 777	84 042	47·3	6·66E7	5 900	3·32	71·7	110·4
A PD 3.0	1 988	96 778	48·7	7·67E7	6 650	3·34	71·2	109·7
F RECCE	335	108 885	325·0	1·93E8	6 463	19·29	27·5	
A RECCE 1.0	1 374	54 451	39·6	5·11E7	4 753	3·45	74·6	
A RECCE 2.0	2 648	137 191	51·8	1·37E8	9 529	3·60	69·1	107·8
A RECCE 3.0	2 770	143 535	51·8	1·49E8	10 328	3·73	68·9	106·6

Notes: the human assessment MI survey values are respectively: 1:34·4; 2:40·9; 3:93·5.

MIs are not available for all software baselines due to changes in comment quantity and quality over the life cycle.

Figure 7 shows the history of the IMOM software model MI and depicts a maintainability profile over time. In this figure, software baselines with equivalent functionality have been appropirately positioned with respect to the functionality axis for ease of comparison.

This metrics-assisted analysis provided several interesting results. First, the original FORTRAN baseline became less maintainable over time. However, since no maintainability measurements were in place during this time, there was no objective means for quantifying the impact of each change made to the system.

Next, the translated C baseline of software suffered the same maintainability degradation as the FORTRAN from which it was derived. This is because it not only had exactly the same software design and architecture as the FORTRAN, but the change path was also very similar. This indicates that direct translation of source code from one language to another does little if anything to improve maintainability and may actually make the software less maintainable, as it did in this case.

Additionally, the re-engineered software showed improved maintainability for func-

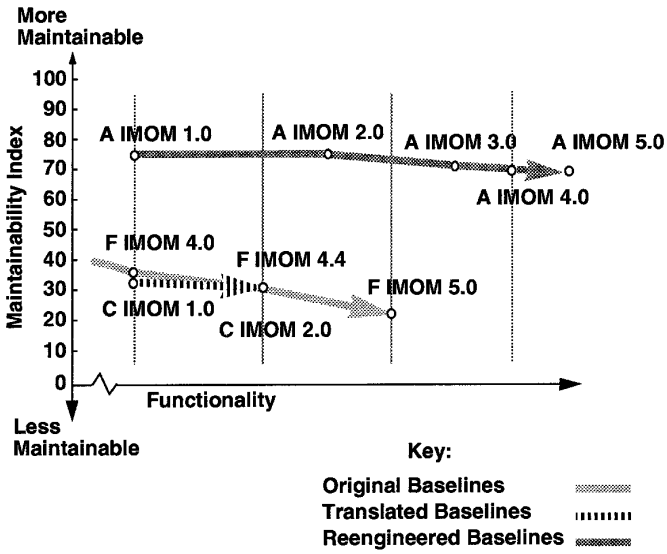

Figure 7. Overall maintenance profiles for the IMOM family

tionally comparable baselines of the software. Predictably, the maintainability of the Ada baseline began to decline, as indicated by the MI for the last measure baseline. After periodic engineering metrics assessment of the software identified decreasing maintainability, a human perspective helped to identify root causes and provided suggestions for improvement from both process and product perspectives. As this case study compares baselines written in three programming languages including Ada with object-orientation, it is important to emphasize that MI usage has shown no apparent bias for or against any particular implementation language.

Finally, it is important to emphasize the human effort and judgement in this process. Metrics alone did not provide sufficient justification for change. Software practitioners, however, were able to use effectively the measurements to guide their thinking and provide an unbiased perspective, while keeping in mind that maintainability, not a high MI value, was the ultimate objective.

In summary, tracking the MI for the released baselines of the IMOM system turned out to be very helpful in making project decisions. Periodic assessments provided new insights into understanding software system maintainability. For example, when the need for re-engineering of software becomes apparent, code translation will not suffice. Additionally, it was determined that the granularity of the measurements over time was still too large. Taking measurements during major version builds is not often enough. In order to be more effective, measurements should be integrated into the software change process such that engineers can easily, routinely, and even automatically calculate maintainability levels before and after each maintenance session. The following case study further illustrates this point, and Section 4 of this paper discusses how such a process might be implemented.

3.4. Case study 3

In one software system being maintained by the INEL, a particular Ada software package contained several subroutines, which were scheduled for functional enhancement. The software maintainers dreaded making changes to that package. In particular, one of the subroutines was implemented via some extremely ugly code. No one was absolutely sure of what the requirements were, no one was sure of the design, no one really understood the implementation. It has been constructed by combining several old algorithms and legacy code segments into a new subroutine. It could be counted on for some types of processing, but outputs for some less common types of inputs could not be easily determined. The maintainers did not dare touch that piece of code for fear of breaking it. The maintainers knew the subroutine was in bad condition, yet it remained in a state of degradation through nearly two years of software changes.

Finally, an MI assessment of the package indicated it was unmaintainable. This was not unexpected, but the difference was that until now the maintainers had had no way of quantifying just how bad the subroutine really was. A request for a suggested enhancement to the package was received. We provide the following case study to illustrate an approach to applying measurement during maintenance as an alternative means for gradual perfective maintenance.

Using the MI as a guide, it was determined that before the functional enhancement should be implemented, the code should be redesigned. After the redesign, the functional changes would then be made to the redesigned component. The maintainers went back to the customer, explicitly redefined the requirements, and redesigned the package and associated subroutines. No functionality was shifted outside the package being redesigned; rather, the internal structure of the package (the decomposition of the package into subroutines) was dramatically changed. Of necessity, functionality was shifted outside the worst subroutine—i.e., the new decomposition changed the overall intent of the subroutine, although a key algorithm remained implemented in that subroutine.

After redesign, the maintainers implemented and tested the new version of the software, and added the enhancements. Tables 5 and 6 show the post-implementation assessment. Table 5 identifies the overall package metrics, while Table 6 identifies the metrics of the primary subroutine within the package.

Several points of interest are indicated by this type of approach. Traditionally, metrics have been used to target modules for perfective maintenance. Often, metrics are determined for a software system, and then the most grievously offending routines are targeted for

Table 5. Perfective maintenance example of an Ada package

Baseline name	Subroutines	Total LOC	Total effort	Total *VG2*	Three-metric MI
Initial	3	663	2 233 072	49	33·55
After redesign	14	732	480 261	64	70·13
After functional enhancements	14	748	499 474	67	69·60

Table 6. Perfective maintenance example of offending subroutine in an Ada package

Baseline name	Subroutine LOC	Subroutine effort	Subroutine *VG2*	Subroutine three-metric MI
Initial	622	2 216 499	45	6·47
After redesign	196	182 216	18	39·93
After functional enhancements	212	201 429	21	37·62

redesign or restructuring. This approach works well if an organization has enough resources to go back and make perfective maintenance changes based on the singular merits of developing higher quality software. However, in most maintenance environments, the software maintainers do not have sufficient resources to use this approach fully.

By using MI analysis to quantify maintainability of subroutines which are already scheduled for change, either to fix defects or to make enhancements, a decision can be made as to the most appropriate type of change to make to the subroutines. For subroutines which have a low maintainability, complete redesign may be appropriate, as described in the example. For subroutines which are moderately maintainable, it may be advantageous to perform some restructuring, either re-engineering or redesign. For subroutines which have high maintainability, it may make sense to just go ahead and make the change. Either way, after modifying the subroutines scheduled for change, a post-implementation assessment should be made to again determine if the code is maintainable, or if further work should be performed.

By integrating MI measurement into the software change process, the overall system maintainability will evolve in an organized fashion. This approach provides the appropriate controls for ensuring minimal system degradation over time. In the best case, the software can actually become more maintainable as changes are made. It is also highly cost effective, since subroutines already targeted for change are the ones being improved. This approach has shown to be an appropriate technique for focusing a metrics application in a specific effort to effectively control software degradation.

3.5. Case study 4

As part of an INEL-sponsored software metrics research study, a relatively new, proprietary, C++ software system was assessed. It was believed that a metrics analysis of this system would provide insight into the structural changes of the system over the maintenance life cycle. The software system in question was less than one year old. Source code was available for two baselines of the system: the initial release of the system and the year old version. The inital release will be referred to as baseline *A* and the current version will be referred to as baseline A^C.

The fact that this system is object-oriented presents special concerns when gathering static code metrics. Traditional code metrics were originally intended for functionally decomposed programs. The following case study serves to illustrate that the MI, despite

its traditional roots, might also be used to quantify maintainability in object-oriented code even if the fit is somewhat less than perfect (see also Case Study 2). Additionally, this study again reminds us that code maintainability decreases over time as a result of change.

There is insufficient evidence from academia and industry to ascertain MI fit for use on object-oriented systems. However, object-oriented systems are primarily comprised of operators and operands, lines of code, lines of comments, and have a number of paths through a module or system as do more traditionally designed systems. Furthermore, the software maintenance practitioner still is interested in and requires the means to measure code density, size, comments and execution logic. Although object-oriented design reflects multiple levels of abstraction within the design, implementation in languages such as C++ continues to possess numerous parallels with traditional approaches. Therefore, existing MI metrics may provide a starting point.

Plews (1993) performed a study which tested the hypothesis that the use of object-oriented programming languages results in less complex software. His conclusions are that pure object-oriented languages (Actor and Smalltalk) show clear improvement for reducing complexity, while object-based or hybrid languages (C++ and Object Pascal) show moderate improvement in reducing complexity and are more reliant on the software engineering skills of the analysts and programmers in order to reduce complexity. Traditional functional languages ('C', Pascal and Modula-2) were rated lower at reducing complexity. In the study conducted by Plews, both Halstead's and McCabe's metrics were used. Statistically significant results were achieved with the pure object-oriented languages. Results were less significant for the hybrid and traditional languages.

Henry and Humphrey (1993) performed a similar, fairly comprehensive study regarding the differences between object-oriented languages and procedural programming languages. Their conclusions are that more maintainable code is written using an object-oriented programming language (Objective-C) than with procedure-oriented languages ('C'). Their study indicated that modifications to object-oriented software required fewer changes to the source code and were more modularly localized by a statistically significant margin. All other cirtiera of the study showed that the object-oriented software was never worse than the traditionally developed software.

The C++ system was analysed with the results shown in Table 7.

By using this knowledge as the basis for applying MI to this system, and comparing it with MI data from other more traditionally structured systems, it is projected to be highly maintainable both at the system and at the subroutine level. Additionally, the extended cyclomatic complexity (*VG*2) is very low. Other studies have suggested that a

Table 7. Software metrics summary for C++ case study

Baseline name	Sub-routines	Total LOC	Ave LOC	Total effort	Total *VG*2	Ave *VG*2	Three-metric MI	Four-metric MI
A	396	9 469	23·9	3 155 684	950	2·4	91·0	121·1
A^C	537	14 703	27·4	7 133 268	1 613	3·0	86·9	116·4

Table 8. Subroutine maintainability index categorization

Baseline name	Total number of subroutines	High maintainability	Moderate maintainability	Low maintainability
A	396	323	59	14
A^C	537	434	67	36

typical functionally decomposed system has subroutines with an average *VG*2 around 12, while an object-oriented system generally has a *VG*2 of around three (Berard, 1992; Martin, 1993, pp. 42–47). This particular system appears to fit with other published data.

In order to get a fuller picture of the system, a synthesis which takes into consideration each of these metrics in relation to each other needs to be determined. This is where the MI metrics provide a valuable addition to understanding the meaning of traditional metrics results. Notice that a comparison between the initial source baseline *A* of the software and the current source baseline A^C shows that the current overall system is slightly less maintainable. Thus, over the course of one year of software maintenance, the software is already beginning to degrade. Table 8 shows the maintainability category of the subroutines, and Figure 8 shows the categories in percentage terms.

In an effort to indicate system maintainability more realistically over time, the metrics for the subroutines which are part of the baselines analysed were compared. There were 29 subroutines in which the MI improved between baselines, the average improvement being 4·0 on the MI scale. The were 204 subroutines which remained the same. There were 155 subroutines in which the MI declined between baselines, the average drop being 8·6 on the MI scale.

As in previous studies, root causes of the declines in maintainability were not readily apparent. However, metric assessment experience would not suggest that the following be examined: (1) increased functional capability of the system; (2) increased application domain complexity; and (3) a maintenance process which fails to consider overall system and module level decomposition factors.

Another perspective on this is illustrated in Figure 9. In this figure, the 155 subroutines where the MI declined during maintenance are graphically compared. The distribution of subroutines where MI has a large negative delta is quite evenly divided across the MI scale. This may be a result of making changes which affect numerous segments of code across the system; therefore, it appears that there may be some undesirable coupling

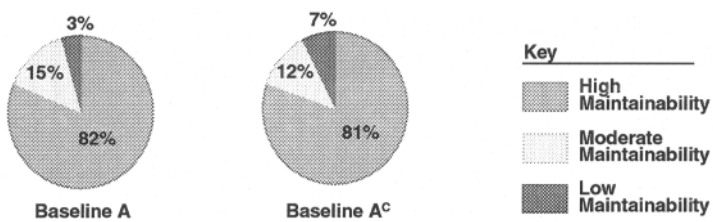

Figure 8. Subroutine maintainability index categorization by percentages

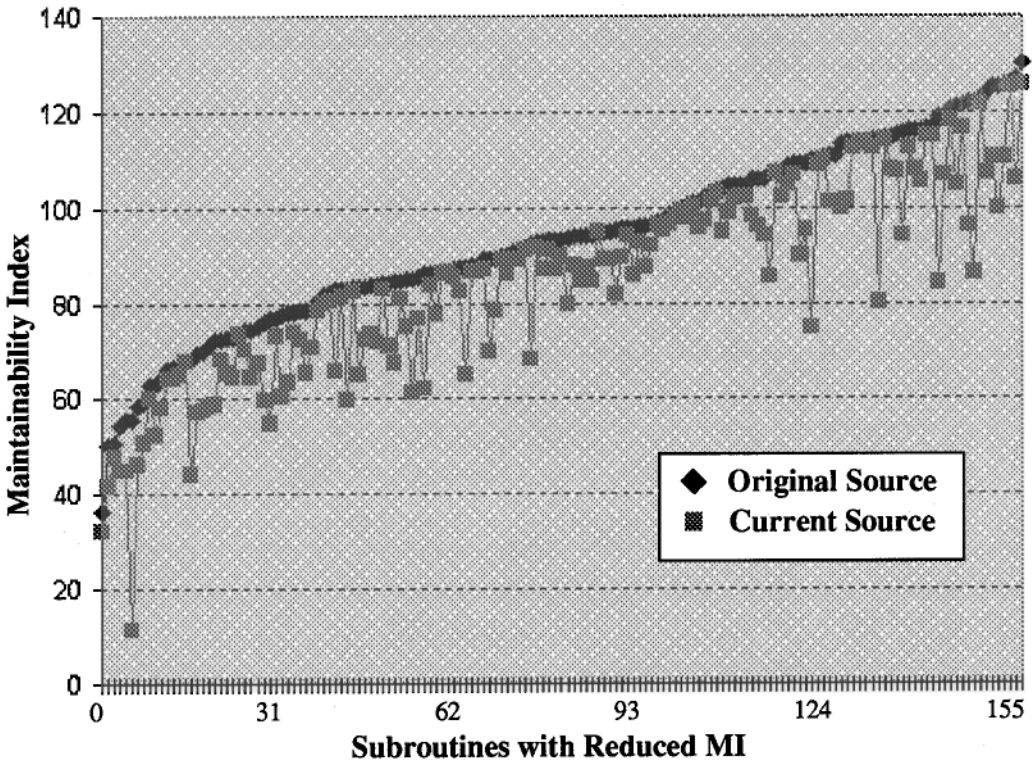

Figure 9. Subroutines that decreased in MI from baseline A to A^c

between subroutines. It may be that when the maintainers implemented a new capability, many subroutines were impacted. Further investigation is required to confirm or refute these implications.

Figure 10 shows the distribution for the 30 subroutines where MI improved. It is quickly apparent that most subroutines changed very little. Also, most of the subroutines that improved where originally in the high maintainability category already.

As mentioned above, the field of object-oriented metrics is a fairly new one. Few metrics have yet been invented, and these have not necessarily been fully validated or tested for usefulness. The software package UX-Metric, however, does calculate some object-oriented metrics on C++ code. The Tables 9 and 10 show the results for baselines A and A^C.

It appears from these metrics that the size and complexity of the average class is growing during maintenance. An increase in the average number of members per class would seem to indicate that increased functionality has been added as a result of adaptive maintenance; this alone would imply that complexity (*VG*2) and size (LOC) have increased (which fits with Table 7). In other words, these results appear to fit with the conclusions gathered from the traditional metrics from the MI analysis. While a new MI metric which incorporates complexity measures specific to object-oriented code should probably be

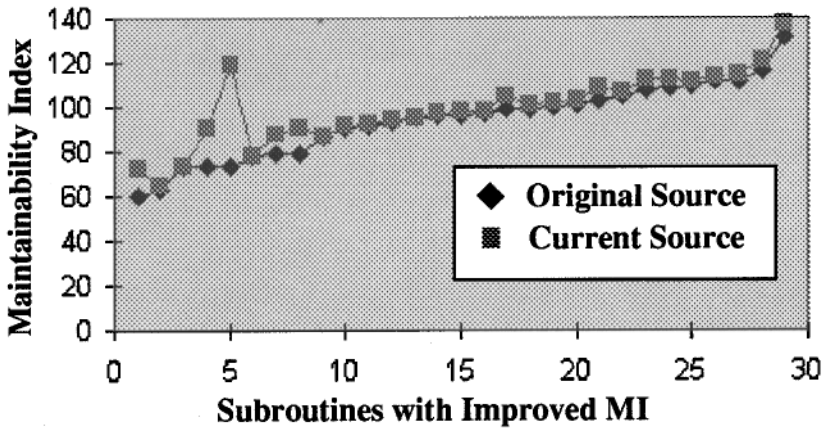

Figure 10. Subroutines that increased in MI from baseline A to A^c

Table 9. Object-oriented metrics for Case Study 4, part 1 of 2

Baseline name	Number of classes	Average members per class	Average private members per class	Average public members per class	Average protected members per class
A	55	28	9	18	0
A^C	55	34	12	21	0

Table 10. Object-oriented metrics for Case Study 4, part 2 of 2

Baseline name	Average inline members/class	Average virtual functions/class	Explicit inline functions	Average friend functions/class	Average friend classes/class
A	11	2	0	0	0
A^C	14	2	0	0	0

derived, the old MI still would seem to be a sufficiently good predictor that it might be used in the interim.

In summary, even though this case study dealt with object-oriented code, we saw that there were similarities to function-oriented code. First, the code maintainability again decreased over time as a result of change. Second, as a minimum, size and complexity are measurements of interest to both types of code. So, even though the fit was less perfect, the MI could still be used to advantage by software practitioners and managers. Again, the point of this case study (as well as the others) was to demonstrate that measurement using MI is a useful tool.

3.6. Case study conclusion

While the MI was used in the preceding case studies as the vehicle for measuring maintainability, it is not necessarily the MI itself that was important, but the fact that some metric was determined and used. The main point of the case studies presented here is that periodic measurements of maintainability can and have been used to advantage by software practitioners and managers alike, and that the more often they are used, the more advantageous is the potential to the users.

We continue with a brief dissertation on the types of tools used in the case studies.

4. INTEGRATING MI INTO MAINTENANCE PRACTICES

4.1. Metrics in maintenance processes

In order to quantify a changing software system, the ability to measure maintainability must be put into the hands of the software practitioners who are changing the code. Measurement should be built into the software development environment such that it is a natural extension of both the implemenation and the maintenance processes. To accomplish this, metrics must be easy to use, available on demand and unobtrusive.

The MI metric is but one example of measurements that may be made; it is expected that most organizations will have a slightly different twist on the metrics important to them. Regardless of the particular metric(s) used the key idea is that if it is used throughout the development and maintenance processes, code maintainability can not only remain higher longer, but might actually be built in early. Of course, the metrics serve as maintainability indicators and still must be applied with common sense and good judgement. Good metrics are not the goal; highly maintainable software is.

4.2. Specific metrics tools

There are numerous metrics tools available in the software marketplace. It is not the purpose of this paper to rate or promote any of these tools, but rather to inform the reader of what tools were used for the studies described in this paper.

The metrics tools which were used to collect the raw metrics from the source code for these studies are UX-Metric and PC-Metric, from SET Laboratories, Inc., PO Box 868, Mulino, OR 97042, U.S.A. The results from these tools can be stored in either a report or a spreadsheet format.

The MI models presented in this paper can be used in conjunction with another tool known as MICalc which was developed for Hewlett-Packard by the Software Engineering Test Laboratory (SETL) at the University of Idaho. MICalc uses the metrics generated by UX/PC-Metric and the approriate MI to generate metrics data such as those shown in Tables 1 to 8 presented earlier.

The INEL also developed a similar MI computation engine known as Poly, which was used for some of the MI calculations presented in this paper.

4.3. Metrics within the maintenance process

As discussed earlier, in order for metrics to be effective, they must be made part of the software practitioners' everyday activities. One way that this has been accomplished on one project at the INEL is to tailor the software maintenance environment to include an easy-to-use MI assessment capability. The INEL uses the Rational Apex product from Rational Software Corporation (2800 San Tomas Expressway, Santa Clara, CA95051, U.S.A.) as one of its software development and maintenance environments. Apex is a comprehensive Ada software development environment/compiler/configuration management product. The Apex environment may be tailored to meet project requirements. The INEL added a 'Run Metrics' option to the Apex tools menu which invokes a Unix shell script with a specified Ada main program, subsystems, directories or individual Ada source code files. The shell script then invokes a metrics analyser (in this case SETL's UX-Metric) to extract the detailed metrics data. These data are then passed on to a program which computes the MIs for the specific source code, and generates a report. The report is then displayed via a report viewer. Thus, with very little effort, the software practitioner or manager gains a tremendous amount of knowledge regarding the subject software, right while it is being developed or modified. Figure 11 diagrams this process.

In practice, a software maintainer or developer can at any time measure the code being worked on using the integrated metrics environment, and use the result to guide source code modification or construction. The software practitioner's user interface perspective is shown in Figure 12.

4.4. Metrics within the development process

When new code is being developed, it is easy to see how having an on-demand metrics assessment capability can lead to the development of more maintainable software. After a module has been designed and coded, its maintainability can be measured. Should the MI evalaution predict maintenance difficulty, the module may be redesigned and coded to achieve better maintainability. Again, achieving a good MI is not the goal; rather, using MI will provide an unbiased second opinion as to the state of the software module, thereby gently reminding and encouraging the software developer to improve software engineering skills which lead to the development of higher quality, more maintainable code.

5. CONCLUSIONS AND SUMMARY

In conjunction with the metrics work described in this paper and as a result of applying metrics to multiple real world software maintenance efforts, some insights have been gained:

Figure 11. Integrated environment of measurement tools

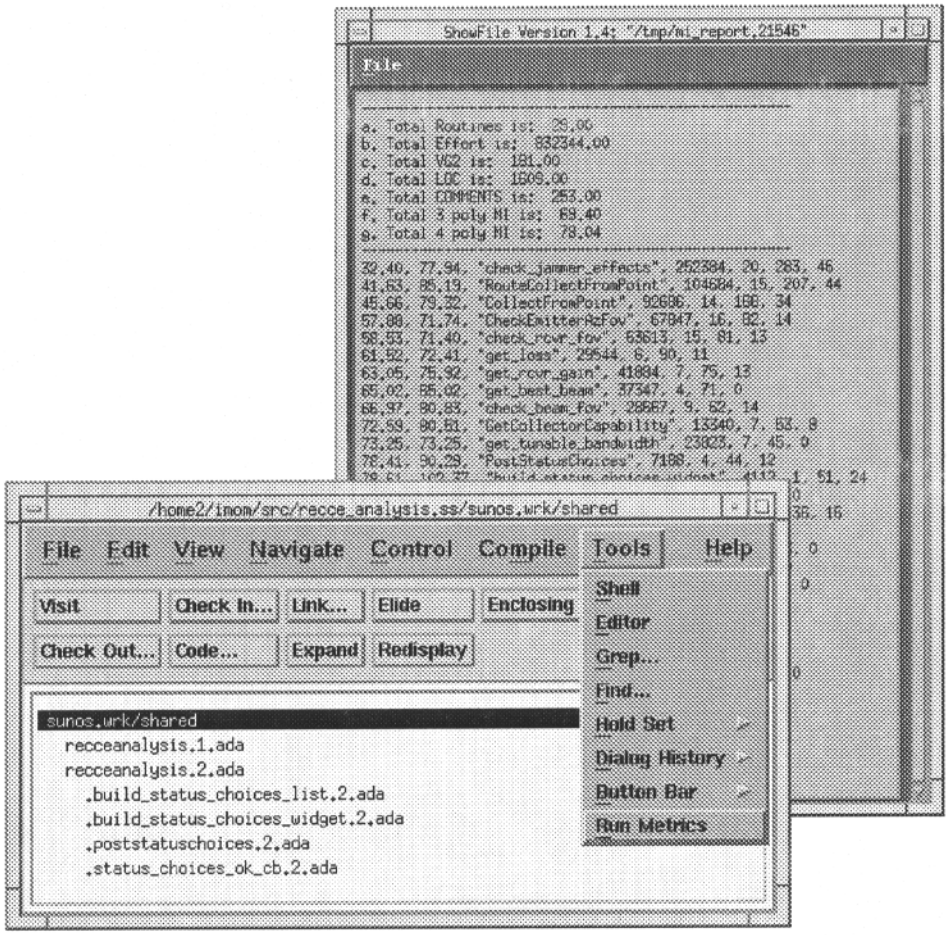

Figure 12. Using metrics tools during maintenance and development

- Metrics assessments are a means to achieving higher software maintainability. They encourage some level of personal accountability when appropiately used in the hands of the maintainers and developers to measure their own efforts. The MI metrics, like all software metrics, can be easily abused when applied out of context. Management and software engineering decisions must retain human insight.
- There is inherent complexity associated with both the work or problem domain being modelled, and the software construction and maintenance processes. An appropriate level of software decomposition for the work or problem domain should be reached without compromising software cohesion. Unnatural decomposition of the work or problem domain in order to achieve a high MI is not appropriate as maintainability could actually be reduced.
- Using the MI metrics increases general software engineering awareness as the metrics become integrated into the software maintenance and development processes and as

its ramifications are better understood by the software practitioners and managers. Numerical excpetion thresholds are generally useful, but must have some flexibility as well. Exceptions to thresholds must be allowed provided there is a solid software engineering justification.

As repeatedly noted and demonstrated in the case studies, code maintainability decreases over time as a function of change. Too often, change, be it corrective, adaptive or perfective, occurs without regard to the effect that it has on the maintainability of the code. As a result, maintainability drops and future changes become increasingly difficult to make. The long-term effect can be described as a downward spiral which culminates in a state where the code eventually becomes unmaintainable. It is our contention that this outcome is at least postponable, and perhaps even avoidable altogether, when code maintainability measurement data are readily available to software practitioners and managers.

From a software engineering perspective, the problem is twofold. First, we must discover a means of measuring the impact of change; this problem was addressed in Section 2.2. Second, the maintenance process itself must be modified such that quantifying maintainability is built in to the process and environment. This means that easy-to-use tools are available to software practitioners and managers. To this end, Section 4 described how 'point-and-click' tools can be easily placed on the software practitioners' desktops.

This paper also discusses the application of software metrics as a tool for quantifying code maintainability and, by extension, for decision making. A minimal set of easily calculated metrics was presented, which when taken together, can produce a single-valued quantification or index of code maintainability. Case studies were presented which not only served to illustrate the degree to which software can degrade over time, but also to illustrate how maintainability metrics, such as the MI, are currently being used in industry to quantify maintainability and aid decision making. Finally, a 'how to' section explains the simple steps involved in providing a tool that software practitioners and managers can use.

The software community at large is gradually becoming aware of the practical application of software metrics. Although a significant amount of work is still to be done, some successes have been achieved. The intent of this paper is to illustrate some of the successes in the hope that they might be emulated; thereby improving maturity levels throughout the industry. Software practitioners and managers can now easily integrate maintainability metrics into the development and maintenance processes.

In summary, using MI to measure system maintainability has been shown to be effective from at least two levels.

- *Periodic software assessment*—using MI to assess system releases over time, and using the resulting information to make strategic system decisions, such as, if re-engineering is in order.
- *Software change integration*—using MI to assess software scheduled for change, and using the resulting information to determine if some redesign or restructuring should take place before making the scheduled change.

The development of large, industrial-size software systems remains difficult at best.

Owing to the difficulty of measuring these highly diverse and complicated software systems, it is no wonder that software metrics have had such a difficult time being accepted into practical application by software developers. However, maintainability metrics, such as the MI metrics described in this paper, have shown a sufficient number of demonstrated successes to warrant practical application in software development and maintenance.

Acknowledgements

We acknowledge and appreciate the helpful comments for the *Journal's* anonymous referees and *Journal* editor Ned Chapin. This work was sponsored by The United States Air Force Information Warfare Center, the Idaho National Engineering Laboratory, the U.S. Department of Energy, the DOE Idaho Field Office under DOE Contract No. DE-AC07-94ID13223, and the University of Idaho Software Engineering Test Laboratory.

References

AFOTEC (1989) *Software Maintainability—Evaluation Guide*, AFOTEC Pamphlet 800-2, Volume 3, HQ Air Force Operational Test and Evaluation Center, Kirtland Air Force Base, NM. 215 pp.

Ash, D., Alderete, J., Yao, L., Oman, P. W. and Lowther, B. (1994) 'Using software maintainability models to track code health', in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 154–160.

Basili, V. R. and Perricone, B. T. (1993) 'Software errors and complexity: an empirical investigation', in Shepperd M. J. (Ed), *Software Engineering Metrics 1: Measures and Validations*, McGraw-Hill Book Company, New York, NY, pp. 168–183.

Belady, L. A. and Lehman, M. M. (1976) 'A model of large program development', *IBM Systems Journal*, **15**(3), 255–252.

Berard, E. (1992) 'Tutorial 30—testing object—oriented software', in *Conference on Object Oriented Programming Systems, Languages, and Applications 1992*, Association for Computing Machinery (ACM), New York, NY. 79 pp.

Booch, G. (1987) *Software Components with Ada*, Benjamin/Cummings Publishing Company, Inc., Menlo Park, CA, pp. 3–5, 554–555.

Coleman, D. (1992) 'Assessing maintainability', in *Proceedings of the Software Engineering Productivity Conference 1992*, Hewlett-Packard, Palo Alto, CA, pp. 525–532.

Coleman, D., Ash, D., Lowther, B. and Oman, P. W. (1994) 'Using metrics to evaluate software system maintainability', *IEEE Computer*, **27**(8), 44–49.

Coleman, D., Lowther, B. and Oman, P. W. (1995) 'The application of software maintainability models on industrial software systems', *Journal of Systems and Software*, **29**(1), 3–16.

Coupal, D. and Robillard, P. N. (1990) 'Factor analysis of source code metrics', *Journal of Systems and Software*, **12**(3), 263–269.

Curtis, B., Sheppard, S. B. and Millman, P. (1993) 'Third-time charm: stronger prediction of programmer performance by software complexity metrics', in Sheppard, M. J. (Ed), *Software Engineering Metrics 1: Measures and Validations*, McGraw-Hill Book Company, New York, NY, pp. 159–167.

Engelhardt, M. (1995) 'Report on statistical analyses of software maintainability indices', an internal report of the Idaho National Engineering Laboratory, Idaho Falls, ID. 33 pp.

Gibson, V. R. and Senn, J. A. (1989) 'System structure and software maintenance performance', *Communications of the ACM*, **32**(3), 347–358.

Halstead, M. H. (1977) *Elements of Software Science*, Elsevier North-Holland, Inc., New York, NY. 127 pp.

Hamer, P. G. and Frewin, G. D. (1993) 'M. H. Halstead's software science: a critical examination', in Shepperd, M. J. (Ed), *Software Engineering Metrics 1: Measures and Validations*, McGraw-Hill Book Company, New York, NY, pp. 184–199.

Harrison, W. (1988) 'Using software metrics to allocate testing resources', *Journal of Management Information Systems*, **4**(4), 93–105.
Harrison, W., Magel, K., Kluczny, R. and DeKock, A. (1982) 'Applying software complexity metrics to program maintenance', *IEEE Computer*, **15**(9), 65–79.
Henry, S. and Humphrey, M. (1993) 'Comparison of an object-oriented programming language to a procedural programming language for effectiveness in program maintenance', *Journal of Object-oriented Programming*, **6**(3), 41–49.
IEEE (1990) *Standard Computer Dictionary*, The Institute of Electrical and Electronics Engineers, Inc., New York, NY, 218pp.
Kafura, D. and Reddy, G. R. (1987) 'The use of software complexity metrics in software maintenance', *Transactions on Software Engineering*, **SE-13**(3), 335–343.
Khoshgoftaar, T. and Oman, P. W. (1994) 'Software metrics: charting the course', *IEEE Computer*, **27**(9), 13–15.
Lehman, M. M. (1980) 'On understanding laws, evolution, and conservation in the large-program life cycle', *Journal of Systems and Software*, **1**(3), 213–221.
Lowther, B. (1993) 'The application of software maintainability metric models on industrial software systems', Master's Thesis, Department of Computer Science, University of Idaho, Moscow, ID. 57 pp.
Martin, J. (1993) *Principles of Object-oriented Analysis and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ. 412 pp.
Myers, G. J. (1977) 'An extension of the cyclomatic measure of program complexity', *ACM SIGPLAN Notices*, **12**(10), 61–64.
McCabe, T. j. (1976) 'A complexity measure', *Transactions on Software Engineering*, **SE-2**(4), 308–320.
Oman, P. W. (1995) 'Applications of an automated source code maintainability index', Technical Report #95-08-SL, Software Engineering Test Laboratory, Universtiy of Idaho, Moscow ID, and presented at the *1995 Software Technology Conference*, Salt Lake City, UT, April 1995, sponsored by Ogden ALC/TISE, Hill AFB in Utah, 27 pp.
Oman, P. W. and Hagemeister, J. (1992) 'Metrics for assessing a software system's maintainability', in *Proceedings of the Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 337–344.
Oman, P. W. and Hagemeister, J. (1994) 'Constructing and testing of polynomials predicting software maintainability', *Journal of Systems and Software*, **24**(3), 251–266.
Oman, P. W., Hagemeister, J. and Ash, D. (1991) 'A definition and taxonomy for software maintainability', Technical Report #91-08-TR, Software Engineering Test Laboratory, University of Idaho, Moscow, ID. 30 pp.
Pearse, T. and Oman, P. W. (1995) 'Maintainability measurements on industrial source code maintenance activities', in *Proceedings of the International Conference on Software Maintenance*, IEEE Computer Society Press, Los Alamitos, CA, pp. 295–303.
Peercy, D. E. (1996) 'Book review of "Improving the Maintainability of Software"', *Journal of Software Maintenance: Research and Practice* **8**(5), 345–356.
Plews, M. (1993) 'Programming language study', *Object Magazine*, **3**(3), 54–55.
Shepperd, M. J. (1993) 'A critique of cyclomatic complexity as a software metric', in Shepperd, M. J. (Ed), *Software Engineering Metrics 1: Measures and Validations*, McGraw-Hill Book Company, New York, NY, pp. 200–213.
Wake, S. and Henry, S. (1988) 'A model based on software quality factors which predicts maintainability', in *Proceedings of the Conference on Software Maintenance—1988*, IEEE Computer Society Press, Los Alamitos, CA, pp. 382–387.
Welker, K. D. (1994) 'Application of software metrics to object-based, reengineered code implemented in Ada', Master's Thesis, Department of Computer Science, University of Idaho, Moscow, ID. 51 pp.
Welker, K. D. and Oman, P. W. (1995) 'Software maintainability metrics models in practice', *CrossTALK: The Journal of Defense Software Engineering*, **8**(11), 19–23, 32.
West, R. (1993) *Improving the Maintainability of Software*, CCTA, HMSO, London. 104 pp.

Zhuo, F. (1992) 'A comparison of software maintainability indices', Master's Thesis, Department of Computer Science, University of Idaho, Moscow, ID. 63 pp.
Zhuo, F., Lowther, B., Oman, P. W. and Hagemeister, J. (1993) 'Constructing and testing software maintainability assessment models' in *Proceedings of the First International Software Metrics Symposium*, IEEE Computer Society Press, Los Alamitos, CA, pp. 61–70.

Authors' biographies:

Kurt D. Welker is a staff software engineer employed by Lockheed Martin Idaho Technologies Company at the Idaho National Engineering Laboratory (INEL). His professional interests include systems analysis, software engineering, object-oriented analysis and design, re-engineering and applied software metrics. Mr. Welker has led a software metrics research initiative at the INEL over the past several years and has over 10 years of software engineering experience. He earned a B.S. degree in Computer Science from Brigham Young University and an M.S. degree in Computer Science from the University of Idaho. His E-mail address is: wdk@inel.gov.

Paul W. Oman is an Associate Professor of Computer Science at the University of Idaho, and an independent software consultant who specializes in software analysis. His consulting practice includes contract work for several international corporations and U.S. government agencies involved with the construction and maintenance of software for power production, medical instrumentation, environmental control systems, and personnel tracking and utilization. He holds the Hewlett-Packard College of Engineering Research Chair at the University of Idaho, where he is Director of the Software Engineering Test Laboratory. He has a Ph.D in Computer Science from Oregon State University, and is a member of the IEEE, IEEE Computer Society and ACM. His E-mail address is: oman@cs.uidaho.edu.

Gerald G. Atkinson is currently a Research Associate with the Software Engineering Test Laboratory at the University of Idaho College of Engineering, where he is completing his M.S. degree in Computer Science. His professional interests include software engineering, software quality assurance and software measurement. Over the past year he has been involved with analysing, measuring and improving group-oriented software development processes. Mr Atkinson received his B.S. degree from Columbus College in Geogria in 1990 and expects to complete his M.S. degree in the Spring of 1977. His E-mail address is: gatkins@cs.uidaho.edu.

